Structure Splitting and Inheritance

Interner Bericht 2005-7

Götz Lindenmaier

Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe
ISSN 1432-7864
goetz@ipd.info.uni-karlsruhe.de

Abstract

The increasing gap between memory and processor performance drives the research for cache optimizations. Recently research concentrates on optimizing pointer based applications. Structure splitting is an important enabling transformation for optimizations that improve the layout of dynamic data structures. Previous work has shown the potential of structure splitting in runtime optimizations. This paper discusses issues of structure splitting applied to inheritance hierarchies of object oriented languages. Inheritance requires similar layout of compound types to simplify type casts. Structure splitting, in contrast, requires a layout that is tailored for a single type. Therefore compatibility between the split type and its super and sub-types is lost. This issue was not addressed by previous work. We explain several strategies to deal with this type compatibility issue and implement two as a compiler optimization. Our experiments show that a careful choice of the strategy is necessary, as they either increase the overhead for accessing cache-neutral data, or they can not achieve the full possible speed up for cache-critical data. Nevertheless, both approaches show considerable speed ups of our tests.

0.1 Introduction

Research on cache optimizations tries to alleviate the effects of the increasing gap between memory and processor performance. Where instruction cache performance reaches a satisfying level, data cache performance still leaves adequate room for improvements. Past years have seen many efforts to improve cache performance of array based programs. More recently research concentrates on optimizing pointer based applications.

This paper considers structure splitting, a fundamental enabling transformation for cache-optimizing pointer based applications. We explore how to apply structure splitting in type systems with inheritance. We discuss the arising problems and present two algorithms with different merits.

Unfortunately it is more difficult to capture the cache deficiencies of pointer based programs than those of array based ones automatically. An analysis can not give an account of the size of a data structure constructed during runtime by looking at the declarations of types or root elements. This is only possible by elaborate analyses or by using profiling data. Further alias problems complicate the correct application of layout transformations, but if a language reduces such problems it is feasible to apply them. Most research for pointer based programs focuses on prefetching. Adding prefetch instructions is always correct – in worst case a badly placed prefetch reduces the cache performance of a program. But prefetching only hides the cost of cache misses instead of avoiding them. Layout transformations, in contrast, reduce the number of cache misses if applied properly.

Structure splitting[CDL99, FK98] is a layout transformation that splits a compound data structure into two (or several) parts. The goal is to obtain a smaller, *hot* structure that contains the cache critical parts, and to place this part preferential in the cache. The other, *cold*, part can be reached by an extra indirection from the hot part or by other mechanisms.

Structure splitting is an important enabling transformation needed to cache-optimize pointer programs. Separating the cold and hot parts in a structure allows to arrange memory regions that only contain hot parts thereby increasing cache reuse. Structure splitting is only effective if the positive effects from placing the hot part cache conscious outweigh the extra memory consumption and computation effort caused by managing the split cold part.

We achieve structure splitting by parting the type that describes the compound data structure into two. In addition we must adapt all allocations, deallocations and accesses to entities of the type to handle the two new types.

Inheritance poses several additional problems to structure splitting. Efficient implementations of inheritance require a compatible layout of super- and subtypes. Splitting a single type in an inheritance hierarchy breaks the compatibility. The program must perform a field access differently depending on the dynamic type of a data structure.

Splitting the subtypes of a split type accordingly establishes type compatibility between these types. This can make sense beyond guaranteeing compatibility. The hot part of the type is inherited to all subtypes. If this part

also is hot accessing instances of the subtype the splitting will allow further performance gains.

Super types of a split type can be treated as super types of the hot part or as super types of the cold part. The first possibility allows full type compatibility, but obscures the cache performance as the hot part increases in size by adding potential cold, inherited fields. Inheriting the super types to the cold part requires a dynamic type cast. Whenever the program casts an instance of the split type to one of it's super types, it must replace the reference pointing to the hot part by a reference pointing to the cold part. This causes additional computations. A third possibility is to apply a "fake" split to the super types: They are split with an empty hot set. This must be applied to all further subtypes, too, and therefore causes additional indirections for all accesses.

We implement the proposed optimizations as true compiler optimizations, i.e., based on a high level intermediate representation. This reduces the overhead caused by the splitting compared to a source to source transformation.

The next section explores the problems and possible solutions of structure splitting with inheritance in detail. This Section gives algorithms to apply structure splitting. Section 0.3 explains type clustering, an optimization that profits from structure splitting. In Section 0.4 we introduce our implementation of type clustering. Finally we present experimental numbers to evaluate the different proposed solutions.

0.2 Structure Splitting and Inheritance

0.2.1 Premises: A Type Hierarchy

Given a program P and a set of types consisting of compound types and primitive types.

A compound type is described by its name and a list of members. The type of a member, the type it belongs to and further attributes describe a member. We call the type a member belongs to the *owner* of the member.

A member is hot if it belongs to a type of which many instances are allocated and if it is accessed frequently during runtime. All other members are cold.

The types form a directed, acyclic inheritance graph. The edges in the graph are the inheritance relations. Each type inherits from a set of other types. If type A inherits from type B, we say A is *subtype* of B and B is *super type* of A.

A member can *overwrite* a member of a direct super type. An instance of a compound type comprises all members specified in the type and all members of the super types of the type if they are not overwritten by a member of the type. This expresses the inheritance of members from the super types. Replicating the members in the subtype and installing an according overwrites relation makes the inheritance explicit. We mark such replicated members as *inherited*. See Fig. 1 for an example.

Given these definitions the type and inheritance system is described by three relations: the inheritance relation, the owner-member relation and the

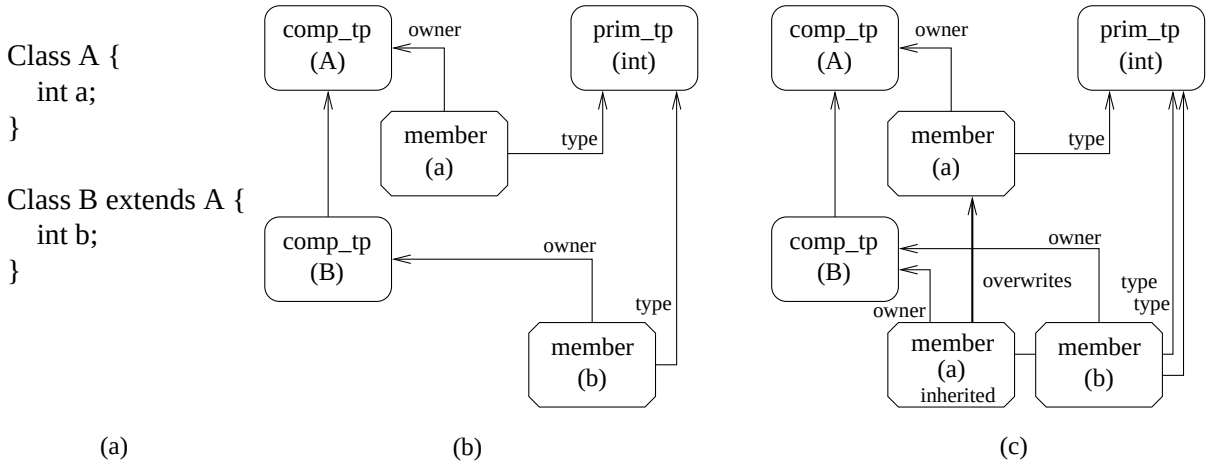

Figure 1: Inheritance representation. (a) shows a declaration of types in pseudo-code. (b) shows the straight forward representation. Member a is inherited to B implicitly. (c) shows the representation of explicit inheritance using the overwrites relation.

overwrites-relation.

All accesses to members are explicit in P. This means each access specifies explicitly the member accessed. During runtime an access either accesses this member or a member that overwrites this member. The compiler generates the proper access mechanism from the specification of this member. [Lin02] describes an implementation of this type representation in the intermediate representation Firm.

Finally all members are annotated with an access statistic that gives an estimate over the number of accesses to this member during runtime. This article does not deal with ways to compute this estimate.

0.2.2 Goal of the Optimization

Given a program P, a fixed type T in P and a subset $M_{T,hot}$ of the members M_T of T. $M_{T,hot}$ contains only hot members. None of the members in $M_{T,hot}$ overwrites a member in a super type of T.

We want to split T into two parts so that one contains all members from $M_{T,hot}$, and the other contains the members $M_{T,cold} = M_T \setminus M_{T,hot}$. Effects on P should be minimal with respect to additional computations and memory consumption.

0.2.3 Structure Splitting

Structure splitting parts T into two types T_{hot} and T_{cold}. T_{hot} emanates directly from T by removing all members of T that are in $M_{T,cold}$. We add the members

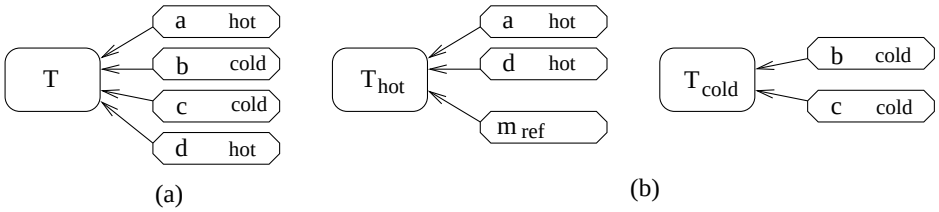

Figure 2: Splitting of a single type T: (a) shows the original type, (b) shows the split type.

of $M_{T,cold}$ to a new type T_{cold}. Further we generate a new member m_{ref} that can hold a reference to an instance of T_{cold} and add m_{ref} to T_{hot} (Fig. 2).

We replace all allocations of an instance of T by two allocations of instances of T_{hot} and T_{cold}. After these allocations we add code that stores the address of the instance of T_{cold} in the m_{ref} field of the instance of T_{hot}. The allocation returns the reference of the instance of T_{hot}. Further we adapt all references to members of T_{cold} to first follow the indirection m_{ref}. We change deallocations so that the instance of T_{cold} and than that of T_{hot} are freed.

0.2.4 Structure Splitting and Subtypes

The members of T are inherited to or overwritten by its suBTypes BT_i. To guarantee that type conversions between T (respectively T_{hot} and T_{cold}) and its subtypes still function we split each BT into BT_{hot} and BT_{cold}. Member sets $M_{BT,hot}$ and $M_{BT,cold}$ initially contain the members that overwrite members in $M_{T,hot}$ and $M_{T,cold}$, respectively. The relations (T_{hot}, BT_{hot}) and (T_{cold}, BT_{cold}) are added to the inheritance relation (Fig. 3). We adapt the code for BT as described for T. Finally we do the same for all subtypes of BT.

If an $m \in M_{T,hot}$ is inherited to BT we have no knowledge about the accesses to m in its role as a part of a BT instance. We must assume that it is also hot so that placing BT_{hot} preferred in the cache delivers a performance gain. Else splitting BT only adds further costs.

Each subtype may add new members to T. We must decide whether these members remain in BT_{hot} or whether we move them to BT_{cold}. If a new member is hot, we expect a performance gain from splitting BT and leave all new hot members of BT in BT_{hot}. We move all new, cold members to BT_{cold}. If this increases the size of BT_{hot} materially we should reevaluate the hot set we started off with by defining stronger thresholds.

0.2.5 Structure Splitting and Super Types

Super types of T have no hot members so that splitting a super type delivers no performance gain. Else choose T to be this super type.

T inherits members from its suPer Types PT_i. If we inherit these members to T_{hot} after splitting (adding (PT, T_{hot}) to the inheritance relation) T_{hot} contains

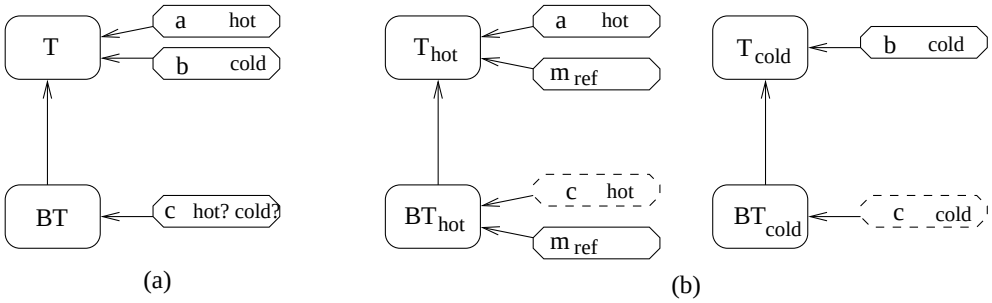

Figure 3: Splitting of a subtype: (a) shows the original type, (b) shows the split type. Field a is inherited to BT_{hot}. Field b is inherited to BT_{cold}.

cold fields. Casting instances of T_{hot} to super types works without additional hassle but we increase T_{hot} unnecessarily, eventually destroying the possible performance gains. We call this strategy *type safe structure splitting*.

Alternatively we can inherit these members to T_{cold} after splitting (adding (PT, T_{cold}) to the inheritance relation). In this case T_{hot} has the intended small extent. Unfortunately the access of members of PT now depends on the dynamic type of the accessed entity. If it is a 'real' instance of PT we can directly access its members. If it is an instance of T cast to PT (i.e., the reference addresses T_{hot}) we need to follow the indirection m_{ref} first.

There exist two solutions to this problem.

First we can split the super type into PT_{hot} and PT_{cold}, where PT_{hot} is a dummy that only contains the member m_{ref}. This achieves the exact layout for T_{hot}/T_{cold} as described above. As a consequence it is necessary to split all other types in the connected component containing T in the inheritance graph. For languages, that define a common super type all types inherit from, this means that all types are split. Then all field accesses pay the penalty of following the indirection. Further this increases the memory requirement of each type.

The better solution is to insert an explicit cast whenever a reference is casted from T to PT or vice versa. After the transformation each cast of a T_{hot} to a PT dereferences m_{ref} so that the reference points to an instance of T_{cold} that is type compatible with PT. Each cast of a PT to a T instance before the transformation must now reestablish the reference to T_{hot}. For this we needed another reference field $m_{backref}$ added to T_{cold} and initialized with the address of T_{hot} after allocation. This additional memory consumption is not cache critical as the new field is added to the cold part of the split type. We call this strategy *optimal structure splitting*.

Adding explicit casts only causes overhead in the order of magnitude of casts, not in the order of magnitude of accesses. But adding the casts requires knowledge of high level type information of reference values when we apply structure splitting. Program P must guarantee type safety. The front end must add a representation of high level type information as resulting from the type

analysis in the semantical analysis to the intermediate representation used for optimization.

If T explicitly overwrites members of PT, we adapt $M_{T,hot}$ and $M_{T,cold}$ according to the strategy before splitting T.

0.2.6 Handling Method Calling

Programming languages utilizing inheritance usually offer polymorphy. This and other mechanisms require static allocated information (the dispatch table) that is accessed depending on the dynamic type of an instance. Compilers typically achieve this by adding a reference m_{static} to the dynamically allocated part of a type that is initialized with the constant reference to the static part.

This reference typically is hot, as it is used in each polymorphic method call. If super types are inherited to the hot split type we can keep m_{static} in the hot part T_{hot}. If super types are inherited to the cold part it is necessary to move m_{static} to the cold part. To avoid the double indirection for the split types we can replicate m_{static} in the hot part at the cost of another field.

0.2.7 Further Problems

T can be an interface to parts of the program P that are not compiled at the same time as T. In this case we can not adapt the allocations, accesses and casts in the other parts of P after deciding to split T.

We can avoid this by using encapsulation for all external interfaces of T. This is only feasible if external interfaces are seldom so that the additional overhead is negligible. If we can not utilize a functional interface an analysis must find out whether the split type is visible out of the compilation unit before applying structure splitting. A runtime optimizations environment renders this issue obsolete as during runtime all parts of the program are known.

0.2.8 Implementation as True Compiler Optimization

A programmer or a tool can apply structure splitting as a source to source transformation([CDL99]). A source to source transformation can easily apply type safe structure splitting. No complex analyses are necessary to find accesses to fields of the split type. Performing optimal type splitting requires a type analyses to find the places where the explicit type casts must be added.

If structure splitting is implemented as a source to source transformation the compiler adds administrative overhead for both parts of the split object. In Java, for example, both classes resulting from a split will inherit from the top class Object. We implemented our optimization as a true compiler pass reducing this overhead.

0.3 Type Clustering

We use a technique we call type clustering to improve the cache performance of pointer based applications. Type clustering is a coarse grained heuristic based on the work by [GTZ98]. Type clustering places all instances of a cache sensitive data type consecutively in memory, e.g, on a special heap. This causes that frequently used instances are more likely to fall into common cache lines. Further frequently used and seldom used instances are separated in memory. This statistically reduces replacement of frequently used cache lines. Finally type clustering can reduce page faults.

Type clustering profits materially from structure splitting. The type cluster shall only contain hot memory locations. If a type has hot and cold fields the cold fields are moved to the cache along with the hot ones if they fall into a common cache line. They increase the overall size of the cluster without significantly increasing the overall number of accesses to the cluster. This contradicts the goal of creating a hot memory region. After applying structure splitting we only apply type clustering to the part with the hot fields. This rules out the negative effects of cold fields in the cluster.

0.4 Implementation

We implemented our experimental framework using the compiler infrastructure CRS [CRS02]. CRS employs the intermediate representation Firm [TLB99]. Firm allows to keep a high level representation of the compiled source language type system. The representation complies to the premises made in this paper – especially are all address computations explicit. This allows easy application of the algorithms proposed here. The compiler we configured for our experiment uses a Java front end and generates a binary executable. The translation does not comply fully with Java standards. The compiler does not support features as dynamic class loading or the full range of reflection functionality. This is sufficient to show the merits and pitfalls of structure splitting in type systems with inheritance.

0.5 Experiments

We tested our implementation with a program that constructs a binary tree and iterates over it. The program constructs the tree and fills it with data in a first iteration. The height of the tree defines the problem size. A second iteration simulates the search for a node by walking the whole tree and comparing an argument with a key in each node.

The program defines four types: A small basic type PT contains two fields. A subtype T of this type implements the tree data structure, and adds the key and several further fields. This type again has two subtypes, BT_1 and BT_2, that each add two more fields. We mark the two fields referring to sons in the tree and the key as hot fields.

Table 1: Cache performance of allocation. Numbers for problem size 20. Numbers for problem size 21 are similar.

Allocation	Accesses		Misses 1. level cache			Misses 2. cache	
	absolute	relative	absolute	relative	rate	relative	rate
no optimization	86061314	100 %	984187	100 %	1.14 %	100 %	1.14 %
type safe	103891382	120 %	1047598	106 %	1.01 %	106 %	1.01 %
optimal	110193306	128 %	1181331	120 %	1.07 %	120 %	1.07 %

We compiled three versions of the program. The first performs no cache optimization. The second performs type-safe structure splitting, the third optimal structure splitting which requires dynamic casts. The dispatch pointer m_{static} is placed in the hot part. The second and third version also perform type clustering. We executed the programs with a cache simulation and measured their run times for two problem sizes (20 and 21).

0.5.1 Cache Effects

First we look at the effects of the optimization on the number of memory accesses and cache misses. For this we employ the cache simulation `cachegrind` which relies on `valgrind` [SN03]. `cachegrind` instruments all memory accesses and simulates an adjustable cache hierarchy with dynamic addresses. The cache hierarchy we simulate has a 64 KByte, 2-way associative first level data cache and a 256 KByte, 8-way associative unified second level cache. Cache lines hold 64 Bytes of data.

Table 1 shows the number of accesses (both write and read) performed during the allocation of the tree. Splitting the objects increases the number of accesses by 20 %. The initialization of the m_{ref} field and the indirection through this field for initialization of the split part account for these accesses. Optimal type splitting increases the number of accesses by an other 8 %. The initialization of the duplicated dispatch reference and the $m_{backref}$ field account for this overhead. Further the initialization of the fields of type B performs a dynamic type cast.

The misses increase less than the number of accesses as the additional accesses exploit reuse. The misses increase according to the increase of the size of the data structure. Type safe splitting adds 1 field and 6.5 % of misses. Optimal splitting adds two more fields and another 2*6.5 = 13 % of misses. The allocation walks sequentially through memory and accesses all cache lines a first time. Therefore all misses are compulsory and the miss rate depends primarily on the cache line size and the reuse in the program, not on the cache size. Therefore the rates are the same for both caches. The additional accesses of type safe splitting have high reuse (m_{ref} is accessed frequently) reducing the miss rate significantly.

Table 2: Cache performance of iteration. Numbers for problem size 20. Numbers for problem size 21 are similar.

Iteration	Accesses		Misses 1. level cache			Misses 2. cache	
	absolute	relative	absolute	relative	rate	relative	rate
no optimization	18350056	100 %	998282	100 %	5.44 %	100 %	5.43 %
type safe	18350056	100 %	532781	53 %	2.90 %	53 %	2.89 %
optimal	18350056	100 %	333515	33 %	1.82 %	33 %	1.81 %

Table 2 shows the corresponding numbers for the iteration. As the iteration uses only fields in the hot part the optimization does not increase the number of accesses. But it reduces the number of misses significantly. As a tree grows exponentially the program contains few locality that can be exploited by the second level cache so that we see similar miss rates for both caches.

The miss rate of the iteration is higher than that of the allocation, as it only accesses the hot fields. The data layout for the unoptimized program mixes hot and cold fields severely increasing the miss rate. The optimization clearly improves the memory performance by separating the hot fields.

The simulation does not capture effects of type clustering on page faults. Further experiments with the same program and only one optimization, structure splitting or type clustering, show now benefits of the individual optimization.

0.5.2 Runtime Effects

Here we look at the run times of the three programs. The runtime is measured with a function of the CRS runtime system that maps to the C function `clock()`. We executed the program on a 1400 MHz Athlon 1800+ with a 256 KB first level cache running RedHat Linux 8.0.

Table 3 shows the runtimes of the three programs. As expected the optimization increases the cost of the allocation pass and reduces the cost of the iteration. The reduction of the iteration runtime is significant - we measure a speed ups of 1.75 to 2.63. In addition, optimal splitting shows a significant speedup over type safe splitting. Further the larger problem size shows a larger speedup. As the simulation gave similar numbers for the cache effects this is either due to the usage of physical addresses in the cache or due to paging effects.

0.5.3 Evaluation

We measured significant improvements by our optimization. This was to expect according to existing literature [CDL99, FK98]. The performance differences

Table 3: Runtime of optimized programs in milliseconds for problem sizes 20 and 21.

Runtime in ms	Allocation				Iteration			
	absolute		relative		absolute		relative	
	20	21	20	21	20	21	20	21
no optimization	420	830	100 %	100 %	210	420	100 %	100 %
type safe	450	890	107 %	107 %	120	230	57 %	55 %
optimal	460	930	110 %	112 %	90	160	43 %	38 %

between the two versions of the optimization show the significance of considering inheritance for this optimization. Different handling of inheritance has significant effects. Optimal splitting shows an advantage over type safe splitting when iterating over the data structure. On the other side optimal splitting requires additional overhead during the allocations.

As this is a constructed test program the costs for the allocation and iteration should not be added to overall evaluate the optimization. The difference between the two optimizations and the cost of the allocation show, that it is necessary to apply the optimization carefully. An analyses that drives the optimization must compare the time a program spends in allocation and accesses to cold fields with accesses to hot fields. It must also consider the number of casts to super types. Depending on these measures it can apply type safe or optimal structure splitting.

0.6 Related Work

In the following we summarize research about cache optimizations of pointer based applications.

Structure splitting is a transformation that has been subject to previous research. [CDL99] perform structure splitting for Java classes. They split classes based on profiling information. Their transformation is performed on source code. The split, cold class inherits from the Java top class Object. This resembles our type safe splitting strategy, but experiences the draw-backs of a source to source transformation discussed above. They do not discuss the problems arising from super or subclasses.

[FK98] propose a variation of structure splitting that reduces the overhead caused by the reference field. They split a type into several parts of equal size, where the first part contains the hot fields. They place the parts at a constant, large offset in memory. The offset affects that the hot parts are placed adjoint in memory. An access to the split part must only add an offset to the reference – far cheaper than following an indirection.

We easily can adapt our implementation of structure splitting for inheritance

to use the allocation strategy of [FK98]. A type cast then adds or subtracts an offset to the casted reference. Such a type cast is probably cheaper than dereferencing fields. We did not yet implement this variation.

Field reordering reorganizes a data structure so that frequently accessed fields lie consecutively in memory. This intends to increase spatial locality. Field reordering is a weak variation of structure splitting useful if splitting is impossible. [CDL99] describe field reordering and propose bbcache, a tool that profiles C programs and proposes an according reorganization of fields of structures.

Several efforts exist that try to improve the cache performance of hot data structures. All these efforts exploit the feature that dynamic data structures can be allocated anywhere without effect on the program semantics.

Clustering tries to place several instances that are used shortly after each other in a common cache line. Clustering is a fine grain allocation scheme.

Coloring organizes the allocation so that a dedicated subset of all instances, independent of their type, are mapped to an exclusive part of the cache. This guarantees that the data in this set is not evicted from the cache by accesses of the rest of the program.

Type clustering as we use it differs from clustering and coloring as it places all instance of a certain type in a dedicated memory region.

[CHL99] propose the tools ccmorph and ccmalloc. These tools are linked to C programs. Calls inserted in a program reorganize the data structures made known to the tools according to the strategies clustering and coloring.

[CL98] use copying garbage collection to reorganize a data structure cache conscious. They collect profiling information during program execution that specifies which instances are used together. The copying phase then places such objects next to each other establishing clustering. The approach has several advantages: It can exploit profiling information of the actual data input. Further the reorganization phase is relatively cheap, as the mere copying cost can not be accounted for the optimization. [CHL00] subsumes the research described in [CHL99], [CL98] and [CDL99].

Other optimizations for pointer based applications examine prefetching.

[LSKR95] address pointer and call intensive programs. Their simple and cheap heuristic assumes that instance references passed into a call will be dereferenced after the call. They prefetch such references before the call. Due to its simplicity this approach is easy to implement as a compiler optimization, but it leaves room for improvement.

[LM99] utilize an analysis to find references that are frequently dereferenced. With this information they prefetch the referenced instance in time. They propose several techniques to increase the time between executing the prefetch and actually accessing the prefetched memory.

[CM01] improve the analysis utilized by [LM99]. Their major achievement is, that they formulate the analyses as an inter procedural data flow analysis. They show how to apply the optimization to Java.

[SAG+01] prefetch objects in linked lists. They recognize iterations through the list by finding induction pointers. They predict the address of the next list

element by assuming that the list elements all lie next to each other in memory. This means that they prefetch a memory location with the address of the actual list element plus an offset.

0.7 Conclusion

We showed how to extend structure splitting to inheritance hierarchies and implemented it as a true compiler optimization. We defined type safe and optimal structure splitting. To apply optimal structure splitting knowledge of high level type information is necessary.

We combined structure splitting with type clustering and executed experiments for this combined optimization. The experiments show significant speed ups for a simple test program. They also show that the way the optimization handles inheritance has major effects on the performance.

Currently we are working on an analyses to drive our optimization. This analyses has to find a trade off between the additional allocation costs, the additional costs through indirections and explicit type casts, and the benefits of accessing hot fields. Depending on the amount of overhead it can apply one of the two versions of the optimization.

Bibliography

[CDL99] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, 1999.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, May 1999.

[CHL00] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making Pointer-Based Data Structures Cache Conscious. *IEEE Computer*, December 2000.

[CL98] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In R. Jones, editor, *First International Symposium on Memory Management*, volume 34 of *ACM SIGPLAN Notices*, pages 37–48, Vancouver, Canada, October 1998. ACM Press.

[CM01] Brendon Cahoon and Kathryn S. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, September 2001.

[CRS02] CRS: Compiler Research System. Website, Fakultät für Informatik, University of Karlsruhe, http://www.info.uni-karlsruhe.de/projects.php/id=56, 2002.

[FK98] Michael Franz and Thomas Kistler. Splitting data objects to increase cache utilization. Technical Report 98-34, Department of Information and Computer Science, University of California, Irvine, October 1998.

[GTZ98] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using sandwich types. In *Proceedings of the 2nd Types in Compilation Workshop*, pages 194–214, Kyoto, Japan, March 1998.

[Lin02] Götz Lindenmaier. libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Interner Bericht 2002-5, Dept. of Computer Science, University of Karlsruhe (TH), September 2002.

[LM99] Chi-Kueng Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2), 1999.

[LSKR95] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: Software Prefetching in Pointer- and Call-Intensive Environments. In *Proceedings of the 28th International Symposium on Microarchitecture*, Ann Arbor, MI, USA, November 1995.

[SAG+01] Artour Stoutchinin, José Nelson Amaral, Guang R. Gao, James C. Dehnert, Suneel Jain, and Alban Douillet. Speculative Prefetching of Induction Pointers. In *Proceedings of International Conference on Compiler Construction 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 289–303, Genova, Italy, April 2001. Springer-Verlag.

[SN03] Julian Seward and Nick Nethercote. Valgrind, version 2.0.0. Technical report, http://developer.kde.org/~sewardj/, 2003.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the Intermediate Representation FIRM. Interner Bericht 1999-14, Dept. of Computer Science, University of Karlsruhe (TH), December 1999.